- Function call: `A=lagmatrix(x,4,2)`
- Output:

$$A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \\ 4 & 6 & 8 & 10 \end{bmatrix}$$

- Function call: `A=lagmatrix(x,2,3)`
- Output:

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

2. Suppose you are told that $1, 3, 5, 7, \ldots$ belongs to class A, and $2, 4, 6, 8, \ldots$ belongs to class B. What information is being carried- what is the underlying process that is producing this data? What does 1.5 produce?

3. While replacing a vector by its coordinate is dimensionality reducing, in some problems it is appropriate to look at expanding the dimension to get what is called a sparse representation. That is, the data that we are presented with is the output of some projection, and we wish to find a higher dimensional representation. Some examples:

   - "Shape from Shading" is where we find a three dimensional model of a shape from a two dimensional representation. For example, find the shape of the clown's face from his photo.
   - "The Cocktail Party Problem". It is a well known phenomenon that, if we are in a large room full of people speaking, we can isolate the sound of the person speaking to us. A particular instance of this problem: Given a tape recording of an orchestra, for example, is it possible to isolate each instrument's sound?

   In terms of what we know from linear algebra, why is this such a harder problem that finding a low dimensional representation?

4. Suppose that some geologists have been studying some different types of rocks in a certain area. Each rock has certain characteristics of interest- for example, color, size, weight, and other compositional items. They give you a non-numerical data set of alphabetic characters, where each symbol represents a characteristic. How do you translate those alphabetical characters into numbers that can be analyzed on a computer[2]?

5. What is the smallest number of basis vectors needed to represent the data:

$$\mathbf{x}_1 = \begin{bmatrix} 2 \\ -1 \\ 2 \\ 0 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} 2 \\ -1 \\ 2 \\ 3 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} -1 \\ \frac{1}{2} \\ -1 \\ 0 \end{bmatrix}, \mathbf{x}_4 = \begin{bmatrix} 6 \\ -3 \\ 6 \\ 9 \end{bmatrix}$$

[2] Caution: Translation into numbers has certain (unintended?) implications- If we just perform a straightforward conversion to the numbers from 1-26, we are bringing along a metric. That is, we are saying that an "A" is closer to "B" than to "Z".

Using this basis, write down the low-dimensional representations for each data point, $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4$.

6. This series of exercises is designed to get you to consider what happens when we put a time series into $\mathbb{R}^2$ or $\mathbb{R}^3$.

   Let $y = \sin(x)$, where x is taken as a sampling of the time interval from $-\pi$ to π. Type the following script in Matlab:

```
x=linspace(-pi,pi,3000);  %Get 3000 samples evenly spaced
y=sin(x);
for j=2:20
  plot(y(1:j-1),y(j:end))
  pause
end
```

   (a) Before beginning, show (graphically) that: $-\cos(t) = \sin\left(t + \frac{\pi}{2}\right)$ and that $-\sin(t) = \sin(t + \pi)$.

   (b) Note that the effect of this code is to plot the series y at time t against the series y at time $t+\tau$. Can you predict what will happen? (HINT: Consider what will happen if $\tau = 0$, $\tau = \frac{\pi}{2}$ and $\tau = \pi$ using the previous part).

   (c) Run the code and write up your observations: What happens to the curve as j changes?

   (d) Try changing y to $\cos(x)$. Any changes?

   (e) Try changing x: `x=linspace(0,100*pi,10000)`, and change y to `y=sin(x)+sin((1/sqrt(2))*x)`, and re-run the code. What happens now?

7. The *dimension* of a set varies with the definition used. We explore a couple of these below:

   - We will define the Euclidean Dimension (D_e) of a set S to be the (smallest) number of basis vectors needed to describe the given set. This is a global quantity, and is the definition we used in linear algebra.
   - The Topological Dimension (D_T) of a set S is something that is local in nature- that is, the topological dimension is something that is determined point by point. To understand this definition, recall that an "$\epsilon-$ neighborhood" (in our usual space R^n) about a point x is the set of points y so that $\|x - y\| < \epsilon$.
     - A set has topological dimension 0 if every point has arbitrarily small neighborhoods whose boundaries do not intersect the set.
     - A set S has topological dimension k if each point in S has arbitrarily small neighborhoods whose boundaries meet S in a set of topological dimension $k-1$, and k is the least nonnegative integer for which this holds.

(a) Suppose S is a circle in the plane. What is D_e, D_T?

(b) Do the same for the sphere ($\|x\| = 1$) in three dimensions.

(c) Suppose S is the letter "X" in the plane. What is D_e, D_T?

(d) Give some examples of sets S where $D_e = D_T$.

(e) Consider the set S in the plane that consists of a filled in square with a line segment attached to one side. Discuss any problems you might have with computing D_T. In determining D_T, be sure to read its definition carefully.

Side Remark: The topological dimension is always an integer (by the way we've defined it). There is also such a thing as a fractal dimension- In this case, the dimension of a set can be a fraction (for example, a dimension of $3/2$ is "more than a line, but does not fill a plane".). This is discussed in a class on fractal geometry.

Part I

Background

Chapter 3

The Theory of Learning

The purpose of this chapter is twofold- To think about *learning* in general, and to get experience with Matlab in programming a reinforcement learning solution to a specific applied problem. Before we get started, let's try to define "learning" by looking at two definitions:

- (From The Random House Dictionary, 1980:) 1. to acquire knowledge (of) or skill (in) by study, instruction or experience. 2. To become informed (of). 3. To memorize (something).
- (From An Introduction to Theories of Learning, B. Hergenhahn, M. Olson, 1997) Learning is a relatively permanent change in behavior or in behavioral potentiality that results from experience and [the change in behavior] cannot be attributed to temporary body states such as those induced by illness, fatigue or drugs.

There is something unsatisfying about both of these definitions. Both would state that simple memorization can be defined as learning. Consider the following examples:

Example 1: In calculus, we learn that the derivative of x^n is nx^{n-1}. It is very common for beginning students to make the following error:

$$\frac{d}{dx}3^x = x3^{x-1}$$

Example 2: In a study of learning in chimpanzees, a game was played. A chimpanzee was presented with two bowls of chocolate candy, one bowl had much more candy than the other. If the chimp pointed at the bowl with more candy, it was promptly given to the *other* chimp. If the chimp pointed to the bowl with the smaller amount, the bowl with the larger amount was given to him.

As it turns out, the chimp could not point to the smaller bowl. He always chose the bowl with more candy (it was always given to the other chimp) and he became very frustrated with the game, voicing great displeasure at having the bowl with more candy removed.

Example 3: You put together a lookup table for your computer. That is, you list a series of possible keyboard inputs and the corresponding response for the computer to make. For example, "When I press the letter L, you will list the contents of the current directory".

In the first example, something was learned, just not the right thing- but the goal is well defined (computing the derivative). In this case, the student has memorized a pattern, but has not realized what the salient items in the pattern referred to (e.g., x^n refers to a variable being raised to a constant power, not simply that we have one thing being raised to a power). In this case, there was an error in distinguishing between inputs (x^3 and 3^x).

The second example illustrates the basic question we must ask in learning problems- Does a solution actually exist? Are we using an algorithm that will pick up that solution? We might hypothesize that the chimp cannot learn to overcome its natural instinct. In machine learning, we need to be sure that we are presenting an association that *can* be learned.

In the third example, has the computer "learned" something? We have changed its behavior, and this change was due to instruction by the programmer. If we told the system to wipe out these commands after 30 days, then the change in behavior would be only relatively permanent. Of course, the authors of the definition had human beings in mind so it may be unfair to apply this to an inanimate object. However, it gives us a starting position in what we should mean when we talk about "Machine Learning", or "Artificial Intelligence". Suppose I were to change the third example:

Example 3: You put together a lookup table for your computer. That is, you list a series of possible keyboard inputs and the corresponding response for the computer to make. For example, "When I press the letter L, list the contents of the current directory". Sometime later, I press the lower case "l", and the computer listed the contents of the directory, *even though it had not been explicitly told to do that.* Now has the computer learned something?

We're in a position now to talk about the learning experience, as it might relate to a machine:

- Learning is initialized with enough samples so that differences can be distinguished, and similarities established. This means that we must use enough samples so that we will adequately represent all the desirable nuances. For example, if I present to you the sequence:

  $$1, 2, \ldots$$

  have I given you enough information to infer the remaining elements? Probably, if I meant that the sequence is $1, 2, 3, 4, 5, \ldots$. Probably not, if I meant the sequence $1, 2, 4, 8, 16, \ldots$, or possibly I meant the sequence $1, 2, 3, 5, 8, 13, \ldots$? Or possibly $1, 2, -1, -2, 3, 4, -3, -4, \ldots$?

  More complexity requires more samples.

  At this point, it would be useful to see how this translates into a mathematical condition. Here is one way to say that a function (what we have been calling a relationship) should not vary wildly-

> Let f be a function over a domain D. The function f is said to be *Lipschitz* if there exists an $M > 0$ so that, for all $a, b \in D$,
>
> $$\mathrm{dist}(f(b), f(a)) \leq M \,\mathrm{dist}(b, a)$$

Exercise: Let $f : \mathbb{R} \to \mathbb{R}$, so that the definition becomes:

$$|f(b) - f(a)| \leq M|b - a|$$

- Show that a Lipschitz function is continuous at any point in its domain. (Therefore, all Lipschitz functions are continuous, but not vice-versa)
- Show that if f is differentiable on its domain and the derivative is bounded, then it is Lipschitz.

So, it is in this sense that having a Lipschitz function means that there is a restriction in how fast it can grow (or vary).

- Input-Output associations should not be so rigorous as to extinguish any possibility of extrapolating to novel stimulus. By this we mean that there is usually a trade off between how accurate a machine can learn, versus how well it can extrapolate. If we force the machine to have zero error during training, it might do very poorly on extrapolation. Furthermore, if the machine always predicts the mean of the outputs, it might be doing pretty well in extrapolation, but is very far away from any particular input-output relationship. An analogy in learning might be this: If a student is punished severely for all errors, it might reduce the possibility of the student answering novel questions in the future; novel stimulus may elicit no response (or a random response, since all incorrect answers get punished equally). On the other hand, if a student is rewarded for any answer that even remotely resembles the correct one, the student may never get an answer that is correct[1].

  This balance between accuracy and extrapolation goes by other names as well: In learning, we might assign it to memorization versus understanding. In statistics, it is known as the trade off between bias and variance.

  In any case, we can say that learning is indexed by appropriate responses to novel input- By appropriate, we will mean that small changes in input characteristics should correspond to small changes in response. Mathematically, again we are talking about a bound on the rate of change.

There are other aspects to learning. For example, we might think of learning as the process of constructing an internalized representation of the outside world,

[1] We have used the terms punishment and reward in a colloquial context- a behavioral psychologist would object to our example, but this is only meant to be a rough analogy to illustrate a point.

and this construction allows us to draw inferences or expectations about the environment. The representation, implicitly referred to in the first item above, begs the question: If input from the environment can be viewed as data streams coming from sensation, how is the information processed?

Patients that have undergone cochlear implants report that initially, sounds are not recognized as hearing- rather, the patients say that they feel strange sensations in their brain. In time, they learn to translate these into recognizable sounds. The point is that we do not hear with our ears; we do not see with our eyes- we hear and see with our brain, which receives the environment as it appears after it has been processed and filtered through the sensory organs. We again ask the question: how is the information processed? If the brain were to receive and try to process every piece of data coming into our senses, we might hypothesize that it would quickly be overwhelmed (think about it- every square millimeter of skin would be constantly sending information at the same time that our ears are picking up sounds, our eyes are picking up images, our nose is picking up smells, we might be thinking about what to have for dinner, etc.). In conclusion, it seems reasonable to assume that each sensation is doing some processing at the signal's point of origin.

When psychologists study learning in babies, they assume that stimulus that has not been internalized elicits attention. For example, studies have shown that even very small babies have developed fairly sophisticated expectations about the world. One study did the following:

The baby is placed in front of a stage. On the stage stands a small screen. A puppet appears from the right side, and is shown to go behind the screen. Another puppet appears from the left side, and again goes behind the screen. In one set of trials, the screen is raised and the two puppets appear. In another set of trials, the screen is raised, and only one puppet appears. In the former case, babies looked briefly at the puppets, but soon began looking elsewhere. In the latter case, the babies stared substantially longer at the single puppet. The babies have begun to understand the concept of addition...?

The conclusion here is that novelty elicits attention, which implies that there is an expectation of what is "normal" based on prior experience.

3.1 Putting it Together: The 3 Phases of Learning

In a broad sense, *learning is the process of building a desirable association between stimulus and response (domain and range), and is measured through resulting behavior on stimulus that has not been previously seen.*

It is important to give a context in which to study machine learning. We will think about learning as a three step process:

1. Building Experience. In hearing this is the processing done at the ear; in sight, this is the processing done at the eye. In mathematics, this might be characterized as getting an understanding of numbers (as a precursor

to Algebra), or of functions (as a precursor to Calculus). In our discussion topic, this is the stage at which we convert alphabetic characters to numerical values. This is where we build the initial tools to distinguish between input patterns that are important, and remove patterns that are not.

2. Building an Internalization.

3. Evaluation of the Internalization. Here we check the internalization that has been built to see if it is robust to changes in the stimulus. Errors here can be coming from either of the first two stages; perhaps the experience base was not broad enough, or the ability to distinguish has not been established.

There could probably be much debate over this kind of categorization, but it is useful as a starting point. In fact, many might argue that points (1) and (2) are essentially the same. We want to distinguish between these for reasons we'll discuss later.

In a machine learning sense at least, the process of learning can be cast in one of two ways:

- Build a function (when the outputs are unique), $y = f(x)$.

- Estimation of a probability distribution (when each input has a whole range of possible outputs), $y = p(x)$, where $p(x) > 0$ for all x, and $\int_{-\infty}^{\infty} p(x)\, dx = 1$

The first model is probably somewhat easier than the second- both in terms of the theory needed, and in the algorithms used for training. We will focus mostly on this first paradigm. The second is studied more in depth in a field known as Reinforcement Learning (in computer science terms, which is different than the psychology term). These two models also distinguish between two different types of learning paradigms: Supervised versus Unsupervised Learning.

In *supervised learning*, we are given examples of proper behavior, and we want the computer to emulate that behavior. In *unsupervised learning*, no desired outputs are given. This might seem to be an impossible task: How can we learn something if we don't know what it is we're learning? Before continuing, let's give some examples of each type:

- **Discovery Learning:** You might have some experience with this. This is where you give a group of students a problem to solve, but do not specify appropriate tools or give examples on how to solve it. This is unsupervised learning.

- Input-output pairs are given, and we wish to find the association. This is supervised learning, since for each input, we're told what the output should be.

- A data set is given, and we'd like to determine *clusters* for the data. That is, we seek a membership function, $m(x)$ where we input a given data point, and output an integer $1, 2, 3, \ldots, k$ where the integer corresponds to the cluster the data is in. In this paradigm, no value of k is given beforehand. This is unsupervised learning, since we don't have any specific goal in mind (number of clusters).

- We can change the last problem to become supervised learning if, for each data point, we also supply a label- for example, each data point belongs to either a "red group", a "blue group", or a "green group".

- **The Mountain Car Problem** Suppose a car is sitting in the valley of a hill. If we apply full throttle, there is not enough power to get us to the top of the hill. If we apply full throttle for a few minutes, then allow gravity to take us down the hill and up the other side, we can start to build momentum by stepping on the throttle when we start to come back up the hill again. Therefore, we must go back and forth across the valley, building up more and more momentum until we have enough power to get us over the top. The problem: For each hillside position, give the sequence of throttles and slides that get us to the top. In this example, we have a goal in mind, but we don't tell the machine how to reach that goal. This again is unsupervised learning.

In general, supervised learning is easier than unsupervised learning in the sense that in unsupervised learning, there will probably be a lot of time "wasted" in trial-and-error where we have to explore the entire input space. In terms of the three processes of learning that was given before:

- In supervised learning, the step of building experience can be separated from the internalization process. We are told explicitly how to differentiate between inputs- we are given some measure of what are the distinguishing characteristics. In this setting, the "expert" sets up the examples of proper behavior.

- In unsupervised learning, these two steps are occurring simultaneously- as we build experience, the internalization is being updated, then we go back to gain more experience to change the internalization again, and the process repeats itself until some goal has been attained. In this setting, we will sometimes get feedback from some "expert". For example, rather than being told that an association is the right association, we might be told that "we're getting warmer or getting colder".

In supervised learning, we can be more specific. The three parts to learning can be recast:

1. The preprocessing phase. In this phase, we are interested in analyzing the desired input-output relationships to see if there are undesirable elements to be removed. That is, we want to find some kind of essential representation of the data. This phase could be doing things like noise removal,

and in general, can be cast as dimensionality reduction. Sometimes this phase is not necessary, but removing redundant information can make the next phase easier- that is, we don't want our learning to become confused. Ideally:

- The compression results in an optimally small set (we will need to construct a measure of what we will mean by "optimal") - small in the sense that the representation carries no redundant information. On the other hand, the resulting compression should also be maximal in the sense that all training data is well represented; we want to be able to maintain an ability to distinguish between distinct stimuli.
- The compression results in a true representation of the original set. If the function f performs the compression, then this will imply that f should be 1-1 on its range, so that it is invertible (on the appropriately restricted domain and range).

2. The learning phase. In this phase, we select an appropriate mathematical model and an appropriate metric so we can measure how well the model is associating. Normally, the model will depend on its *parameters*, and learning is performed via maximization or minimization of the metric. Here we will study some general techniques of optimization theory.

3. The postprocessing phase. Once the associations have been established, it is critical to test the model on new data- is the model generalizing well? It could be that the model is too simple or too complex.

3.2 Exercise Set

1. Consider the concept of *superstition*: This is a belief that one must engage in certain behaviors in order to receive a reinforcement, where in reality, the reinforcement did not depend on those behaviors. Is it possible for a computer to engage in superstitious activity? Discuss in terms of the supervised versus unsupervised learning paradigms.

2. In psychology, an *unconditioned response* is something that is automatic and not something that needs to be learned. For example, salivation with the presence of food, and constriction of the pupil of the eye in the presence of light are unconditioned responses. An *unconditioned stimulus* is the stimulus that elicits an unconditioned response. In classical conditioning, we can train animals to perform a given task by first associating a conditioned stimulus to an unconditioned stimulus. Soon, the conditioned stimulus will elicit the unconditioned response (therefore, the conditioned stimulus becomes a conditioned response). Can such a paradigm be useful in machine learning? Discuss classical conditioning in terms of what we called "learning" in this chapter.

3. **Probabilities**

   A signal light comes on and is followed by one of two other lights. The goal is to predict which of the lights comes on given that the signal light comes on. The experimenter is free to arrange the pattern of the two response lights in any way- for example, one might come on 75% of the time.

   Let E_1, E_2 denote the event that the first (second) light comes on, and let A_1, A_2 denote the prediction that the first (second) light comes on (respectively). Let π be the probability that E_1 occurs.

   (a) If the goal is to maximize your reward through accurate predictions, what should you do in this experiment? Just give a heuristic answer- you do not have to formally justify it.

   (b) How would you program a machine to maximize it's prediction accuracy? Can you state this in mathematical terms?

   (c) What do you think happens with actual subject (human) trials?

4. Harlow's Monkeys (Harry F. Harlow)

   In a test similar to the previous experiment, Harlow had monkeys learn a discrimination task. On each problem, they were to pick one of two objects that had a reinforcer placed under it. There were a total of 344 tasks, each with a different set of objects. In light of the previous problem, how do you think the monkeys performed early on in the trials versus later in the trials?

5. In Jean Piaget's theory of learning, *assimilation* and *accommodation* play a key part. Assimilation refers to a matching between a given cognitive structure and the environment. For example, a cognitive structure defines the boundaries on what a child can learn- if only sucking, looking, reaching and grasping are available to a baby, then experiences are assimilated in those terms. Accommodation is the process by which the cognitive structure is modified. In Calculus I, we learn what a derivative is. Describe this process in terms of assimilation and accommodation.

3.3 A Case Study in Reinforcement Learning

In this section, we will summarize much of what we've been talking about in learning theory in terms of a special case: The problem of the $n-$armed bandit (Barto and Sutton). The name comes from a slang term for a slot machine, also known as a one-armed bandit.

In the $n-$armed bandit problem, we have n slot machines (or equivalently, one machine with $n-$arms), each having a different probability of winning. The goal is to play the slots and to maximize our returns. Before continuing, let us set up some notation: Define

$$Q(a) = \text{ The expected return for playing slot machine } a$$

You can also think of $Q(a)$ as the *mean* of the payoffs for slot machine a. If we knew $Q(a)$ for each machine a, our strategy would be very simple: Play the machine with the largest payoff, and ignore the others. The interesting part of the problem is that we don't know what the returns are! So, we have to estimate the returns. Let

$$Q_t(a) = \text{ Our estimation of } Q(a) \text{ at time } t$$

Our general algorithm should be set up so that our estimates get better over time:

$$\lim_{t\to\infty} Q_t(a) = Q(a) \tag{3.1}$$

Let's start things off with a straightforward estimate. Suppose we play slot machine a a total of n_a times, with payoffs $r_1, \ldots, r_{n_a}$ (note that these values could be negative!). Then we might estimate $Q(a)$ as the mean of these values:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{n_a}}{n_a}$$

In statistical terms, we are using the sample mean to estimate the actual mean. In statistics, the Law of Large Numbers states that as the number of samples increase, the sample mean converges to the population mean. We should also agree on a starting value: By definition, we'll take $Q_0(a) = 0$.

3.3.1 Strategies for the $n-$armed bandit

In this section, we'll describe different strategies one might take to *learn* $Q(a)$.

The Greedy Algorithm

This strategy is straightforward: Always play the slot machine with the largest (estimated) payoff. If a_{t+1} is the machine we'll play at time $t + 1$, then:

$$a_{t+1} = \arg\max \{Q_t(1), Q_t(2), \ldots, Q_t(n)\}$$

where "arg" refers to the argument of the maximum (which is an integer from 1 to n corresponding to the max) . If there is a tie, then choose one of them at random. Let's take a moment to see how we might do this in Matlab. The `find` command will be used to find the maximum values:

```
idx=find(x==max(x))
```

will return all indices of the vector x that are equal to the max. We'll talk about the double equality in class. You should try this command with something like:

```
x=[1 2 3 0 3];
idx=find(x==max(x));
```

and the result will be a vector, `idx`, will contain the values 3 and 5.

Going back to the original strategy, I think you'll see a problem- What if the estimations are wrong? Then its very possible that you'll get stuck on a suboptimal machine. This problem can be dealt with in the following way: Every once in a while, try out the other machines to see what you get. This is what we'll do in the next section.

The $\epsilon-$Greedy Algorithm

In this algorithm we will choose, with probability ϵ, a machine totally at random. Notice that in this case, the number of trials gets larger and larger, $n_a \to \infty$ for *all* machines a, and so we will be guaranteed convergence to the proper estimates of $Q(a)$ for all a machines. On the flip side, because we're always investigating other machines every once in a while, we'll never maximize our returns.

You might note the relationship of what we've just said to Estes' probability matching experiments!

`for j=1 to Number of Trials`

- `Select an action:`
  - `Sometimes choose one at random`
  - `Otherwise, select the action with greatest return. Check for ties, and if there is a tie, pick on of them at random.`
- `Get your payoff`
- `Update the estimates` Q

The first programming problem will be to decide on how to sometimes select a machine at random, and sometimes not. If ϵ =`E` is the probability of this event, and `N` is the number of trials, then one way of selection is to set up an $N-$vector, `greedy`, that will "flag" the events- that is, on trial j, if `greedy(j)`$=1$, choose a machine at random. Otherwise, choose using the greedy method. The following code will do just that (`N` is the number of trials)

```
greedy=zeros(1,N);
if E>0
   m=round(E*N);  %Total number of times we should choose at random
   greedy(1:m)=ones(1,m);
   m=randperm(N); %Randomly permute the vector indices
   greedy=greedy(m);
   clear m
end
```

And here's the full function. We assume that the actual rewards for each of the bandits is given in the vector `Aq`, and that when machine a is played, the sample reward will be chosen from a normal distribution with unit variance and mean `Aq(a)`.

```
function [As,Q,R]=banditE(N,Aq,E)
%FUNCTION [As,Q,R]=banditE(N,Aq,E)
%  Performs the N-armed bandit example using epsilon-greedy
%    strategy.
%  Inputs:
%        N=number of trials total
%        Aq=Actual rewards for each bandit (these are the mean rewards)
%        E=epsilon for epsilon-greedy algorithm
%  Outputs:
%        As=Action selected on trial j, j=1:N
%        Q are the reward estimates
%        R is N x 1, reward at step j, j=1:N

numbandits=length(Aq);       %Number of Bandits
ActNum=zeros(numbandits,1);  %Keep a running sum of the number of times
                             %  each action is selected.
ActVal=zeros(numbandits,1);  %Keep a running sum of the total reward
                             %  obtained for each action.
Q=zeros(1,numbandits);       %Current reward estimates
As=zeros(N,1);               %Storage for action
R=zeros(N,1);                %Storage for averaging reward

%*********************************************************************
%  Set up a flag so we know when to choose at random (using epsilon)
%*********************************************************************
greedy=zeros(1,N);
if E>0
   m=round(E*N);  %Total number of times we should choose at random
   greedy(1:m)=ones(1,m);
   m=randperm(N);
   greedy=greedy(m);
   clear m
end
if E>=1
    error('The epsilon should be between 0 and 1\n');
end
%********************************************************************
%
%  Now we're ready for the main loop
%********************************************************************
for j=1:N
    %STEP ONE:  SELECT AN ACTION  (cQ) , GET THE REWARD  (cR) !
        if greedy(j)>0
            cQ=ceil(rand*numbandits);
            cR=randn+Aq(cQ);
        else
            [val,idx]=find(Q==max(Q));
            m=ceil(rand*length(idx));  %Choose a max at random
            cQ=idx(m);
            cR=randn+Aq(cQ);
```

```
        end
        R(j)=cR;
    %UPDATE FOR NEXT GO AROUND!
        As(j)=cQ;
        ActNum(cQ)=ActNum(cQ)+1;
        ActVal(cQ)=ActVal(cQ)+cR;
        Q(cQ)=ActVal(cQ)/ActNum(cQ);
end
```

Next we'll create a test bed for the routine. We will call the program 2,000 times, and each call will consist of 1,000 plays. We will set the number of bandits to 10, and change the value of ϵ from 0 to 0.01 to 0.1, and see what the average reward per play is over the 1000 plays.

Here's a script file that we'll use to call the `banditE` routine:

```
Ravg=zeros(1000,1);
E=0.1;
for j=1:2000
    m=randn(10,1);
    [As,Q,R]=banditE(1000,m,E);
    Ravg=Ravg+R;
    if mod(j,10)==0
        fprintf('On iterate %d\n',j);
    end
end
Ravg=Ravg./2000;
plot(Ravg);
```

The output of the algorithms are shown in Figure 3.1.

The Softmax Action Selection

In the Softmax action selection algorithm, the idea is to construct a set of probabilities. This set will have the properties that:

- The machine (or arm) giving the highest estimated payoff will have the highest probability.
- We will choose a machine using the probabilities. For example, if the probabilities are $0.5, 0.3, 0.2$ for machines $1, 2, 3$ respectively, then machine 1 would be chosen 50% of the time, machine 2 would be chosen 30% of the time, and the last machine 20% of the time.

Therefore, this algorithm will maintain an exploration of all machines so that we will not get locked onto a suboptimal machine.

Now if we have n machines with estimated payoffs recorded as:

$$Q = [Q_t(1), Q_t(2), \ldots, Q_t(n)]$$

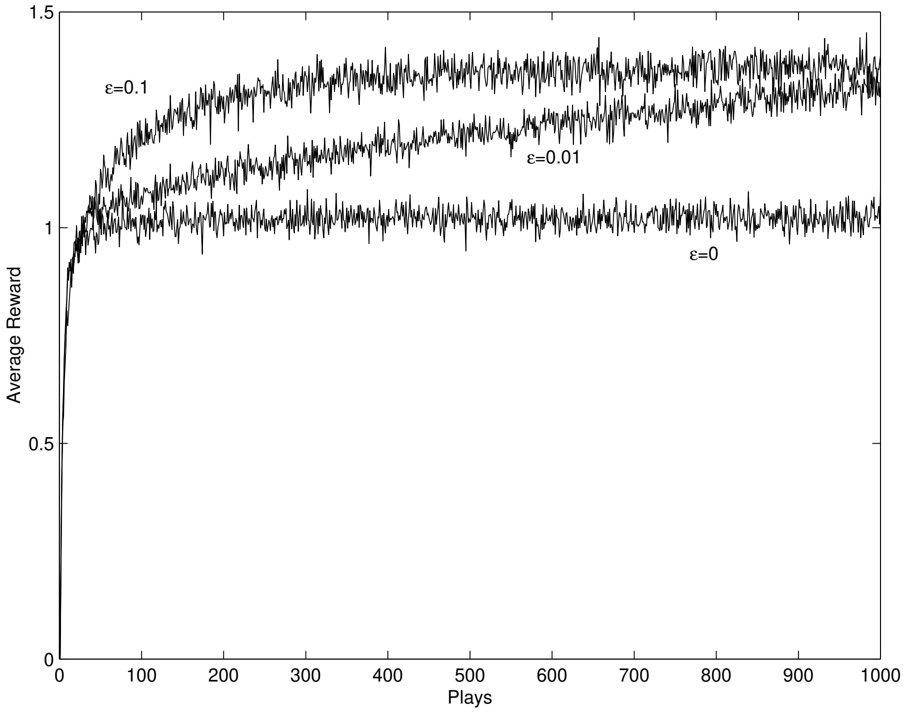

Figure 3.1: Results of the testbed on the 10-armed bandit. Shown are the rewards given per play, averaged over 2000 trials.

we want to construct n probabilities,

$$P = [P_t(1), P_t(2), \ldots, P_t(n)]$$

The requirements for this transformation are:

1. $P_t(k) \geq 0$ for $k = 1, 2, \ldots$ (because all probabilities are positive). Another way to say this is to say that the range of the transformation is nonnegative.

2. If $Q_t(a) < Q_t(b)$, then $P_t(a) < P_t(b)$. That is, the transformation must be strictly increasing for all domain values.

3. Finally, the sum of the probabilities must be 1.

A function that satisfies requirements 1 and 2 is the exponential function. It's range is nonnegative. It maps large negative values (large negative payoffs) to near zero probability, and it is strictly increasing. Up to this point, the transformation is:

$$\hat{P}_t(k) = \mathrm{e}^{Q_t(k)}$$

We need the probabilities to sum to 1, so we normalize the $\hat{P}_t(k)$:

$$P_t(k) = \frac{\hat{P}_t(k)}{\hat{P}_t(1) + \hat{P}_t(2) + \ldots + \hat{P}_t(n)} = \frac{\exp(Q_t(k))}{\sum_{j=1}^{n} \exp(Q_t(j))}$$

This is a popular technique worth remembering- We have what is called a Gibbs (or Boltzmann) distribution. We could stop at this point, but it is convenient to introduce a control parameter τ (sometimes this is referred to as the temperature of the distribution). Our final version of the transformation is given as:

$$P_t(k) = \frac{\exp\left(\frac{Q_t(k)}{\tau}\right)}{\sum_{j=1}^{n} \exp\left(\frac{Q_t(j)}{\tau}\right)}$$

EXERCISE: Suppose we have two probabilities, $P(1)$ and $P(2)$ (we left off the time index since it won't matter in this problem). Furthermore, suppose $P(1) > P(2)$. Compute the limits of $P(1)$ and $P(2)$ as τ goes to zero. Compute the limits as τ goes to infinity (Hint on this part: Use the definition, and divide numerator and denominator by $\exp(Q(1)/\tau)$ before taking the limit).

What we find from the previous exercise is that the effect of large τ (hot temperatures) makes all the probabilities about the same (so we would choose a machine almost at random). The effect of small *tau* (cold temperatures) makes the probability of choosing the best machine almost 1 (like the greedy algorithm).

In Matlab, these probabilities are easy to program. Let `Q` be a vector holding the current estimates of the returns, as before, and let `t=` τ, the temperature. Then we construct a vector of probabilities using the softmax algorithm:

```
P=exp(Q./t);
P=P./sum(P);
```

Programming Comments

1. How to select action a with probability $p(a)$?

   We could do what we did before, and create a vector of choices with those probabilities fixed, but our probabilities change. We can also use the uniform distribution, so that if `x=rand`, and $x \leq p(1)$, use action 1. If $p(1) < x \leq p(1) + p(2)$, choose action 2. If $p(1) + p(2) < x \leq p(1) + p(2) + p(3)$, choose action 3, and so on. There is an easy way to do this, but it is not optimal (in terms of speed). We introduce two new Matlab functions, `cumsum` and `histc`.

   The function `cumsum`, which means *cumulative sum*, takes a vector x as input, and outputs a vector y so that `y=cumsum(x)` creates:

$$y_k = \sum_{n=1}^{k} x_n = x_1 + x_2 + \ldots + x_k$$

For example, if $x = [1, 2, 3, 4, 5]$, then `cumsum(x)` would output $[1, 3, 6, 10, 15]$

The function `histc` (for *histogram count*) has the form: `n=histc(x,y)`, where the vector y is monotonically increasing. The elements of y form "bins" so that $n(k)$ counts the number of values in `x` that fall between the elements `y(k)` (inclusive) and `y(k+1)` (exclusive) in the vector `y`. Try a particular example, like:

```
Bins=[0,1,2];
x=[-2, 0.25, 0.75, 1, 1.3, 2];
N=histc(x, Bins);
```

`Bins` sets up the desired intervals as $[0, 1)$ and $[1, 2)$ and the last value is set up as its own interval, 2. Since -2 is outside of all the intervals, it is not counted. The next two elements of x are inside the first interval, and the next two elements are inside the second interval. Thus, the output of this code fragment is $N = [2, 2, 1]$.

Now in our particular case, we set up the bin edges (intervals) so that they are the cumulative sums. We'll then choose a number between 0 and 1 using the (uniformly) random number $x =$ `rand`, and determine what interval it is in. This will be our action choice:

```
P=[0.3, 0.1, 0.2, 0.4];
BinEdges=[0, cumsum(P)];
x=rand;
Counts=histc(x,BinEdges);
ActionChoice=find(Counts==1);
```

2. We have to change our parameter τ from some initial value τ_{init} (big, so that machines are chosen almost at random) to some small final value, τ_{fin}. There are an infinite number of ways of doing this. For example, a linear change from a value a to a value b in N steps would be the equation of the line going from the point $(1, a)$ to the point (N, b).

   Exercise: Give a formula for the parameter update, τ in terms of the initial value, τ_{init} and the final value, τ_{fin} if we use a linear decrease as t ranges from 1 to N.

   A more popular technique is to use the following formula, which we'll use to update many parameters. Let the initial value of the parameter be given as a, and the final value be given as b. Then the parameter p is computed as:

$$p = a \cdot \left(\frac{b}{a}\right)^{t/N} \tag{3.2}$$

   Note that when $t = 0$, $p = a$ and when $t = N$, $p = b$[2]

[2]In the C/C++ programming language, indices always start with zero, and this is leftover in this update rule. This is not a big issue, and the reader can make the appropriate change to starting with $t = 1$ if desired.

"Win-Stay, Lose-Shift" Strategy

The "Win-Stay, Lose-Shift" strategy discussed in terms of Harlow's monkeys and perhaps the probability matching experiments of Estes might be re-formulated here for the $n-$armed bandit experiment.

In this experiment, we interpret the strategy as: If I'm winning, make the probability of choosing that action stronger. If I'm losing, make the probability of choosing that action weaker. This brings us to the class of *pursuit* methods.

Define a^* to be the winning machine at the moment, i.e.,

$$a^* = \max_a Q_t(a)$$

The idea now is straightforward- Slightly increase the probability of choosing this winning machine, and correspondingly decrease the probability of choosing the others.

Define the probability of choosing machine a as $P(a)$ (or, if you want to explicitly include the time index, $P_t(a)$. Then given the winning machine index as a^*, we update the current probabilities by using a parameter $\beta \in [0, 1]$:

$$P_{t+1}(a^*) = P_t(a^*) + \beta\,[1 - P_t(a^*)]$$

and the rest of the probabilities decrease towards zero:

$$P_{t+1}(a) = P_t(a) + \beta\,[0 - P_t(a)]$$

Exercises with the Pursuit Strategy

1. Suppose we have three probabilities, P_1, P_2, P_3, and P_1 is the unique maximum. Show that, for any $\beta > 0$, the updated values still sum to 1.

2. Using the same values as before, show that, for any $\beta > 0$, the updated values will stay between 0 and 1- that is, If $0 \leq P_i \leq 1$ for all i before the update, then after the update, $0 \leq P_i \leq 1$.

3. Here is one way to deal with a tie (show that the updated values still sum to 1): If there are k machines with a maximum, update each via:

$$P_{t+1} = (1 - \beta) * P_t + \beta/k$$

4. Suppose that for some fixed j, P_j is always a loser (never a max). Show that the update rule guarantees that $P_j \to 0$ as $t \to \infty$. HINT: Show that $P_j(t) = (1 - \beta)^t P_j(0)$

5. Suppose that for some fixed j, P_j is always a winner (with no ties). Show that the update rule guarantees that $P_j \to 1$ as $t \to \infty$.

Matlab Functions `softmax` and `winstay`

Here are functions that will yield the softmax and win-stay, lose-shift strategies. Below each is a driver. Read through them carefully so that you understand what each does. We'll then ask you to put these into Matlab and comment on what you see.

```
function a=softmax(EstQ,tau)
% FUNCTION a=softmax(EstQ, tau)
%   Input:  Estimated payoff values in EstQ (size 1 x N,
%           where N is the number of machines
%           tau - "temperature":  High values- the probs are all
%           close to equal; Low values, becomes "greedy"
%  Output:  The machine that we should play (a number between 1 and N)

if tau==0
    fprintf('Error in the SoftMax program-\n');
    fprintf('Tau must be greater than zero\n');
    a=0;
    return
end

Temp=exp(EstQ./tau);
S1=sum(Temp);
Probs=Temp./S1;  %These are the probabilities we'll use

%Select a machine using the probabilities we just computed.
x=rand;
TotalBins=histc(x,[0,cumsum(Probs)']);
a=find(TotalBins==1);
```

Here is a driver for the softmax algorithm. Note the implementation details (e.g., how the "actual" payoffs are calculated, and what the initial and final parameter values are):

```
%Script file to run the N-armed bandit using the softmax strategy

%Initializations are Here:
NumMachines=10;
ActQ=randn(NumMachines,1);  %10 machines
NumPlay=1000;               %Play 100 times
Initialtau=10;              %Initial tau ("High in beginning")
Endingtau=0.5;
tau=10;
NumPlayed=zeros(NumMachines,1);  %Keep a running sum of the number
                                 %  of times each action is selected
ValPlayed=zeros(NumMachines,1);  %Keep a running sum of the total
```

```
                                 %  reward for each action
EstQ=zeros(NumMachines,1);
PayoffHistory=zeros(NumPlay,1);  %Keep a record of our payoffs

for i=1:NumPlay

    %Pick a machine to play:
    a=softmax(EstQ,tau);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    PayoffHistory(i)=Payoff;
    tau=Initialtau*(Endingtau/Initialtau)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Here is the function implementing the pursuit strategy (or "Win-Stay, Lose-Shift").

```
function [a, P]=winstay(EstQ,P,beta)
% function [a,P]=winstay(EstQ,P,beta)
%  Input:  EstQ, Estimated values of the payoffs
%          P = Probabilities of playing each machine
%          beta= parameter to adjust the probabilities, between 0 and 1
%  Output:  a = Which machine to play
%           P = Probabilities for each machine

[vals,idx]=max(EstQ);
winner=idx(1);  %Index of our "winning" machine

%Update the probabilities.  We need to do P(winner) separately.
NumMachines=length(P);
P(winner)=P(winner)+beta*(1-P(winner));

Temp=1:NumMachines;
Temp(winner)=[];  %Temp now holds the indices of all "losers"
P(Temp)=(1-beta)*P(Temp);

%Probabilities are all updated- Choose machine a w/prob P(a)
x=rand;
```

```
TotalBins=histc(x,[0,cumsum(P)']);
a=find(TotalBins==1);
```

And its corresponding driver is below. Again, be sure to read and understand what each line of the code does:

```
%Script file to run the N-armed bandit using pursuit strategy

%Initializations
NumMachines=10;
ActQ=randn(NumMachines,1);
NumPlay=2000;
Initialbeta=0.01;
Endingbeta=0.001;
beta=Initialbeta;
NumPlayed=zeros(NumMachines,1);
ValPlayed=zeros(NumMachines,1);
EstQ=zeros(NumMachines,1);
Probs=(1/NumMachines)*ones(10,1);

for i=1:NumPlay

    %Pick a machine to play:
    [a,Probs]=winstay(EstQ,Probs,beta);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    beta=Initialbeta*(Endingbeta/Initialbeta)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Homework: Implement these 4 pieces of code into Matlab, and comment on the performance of each. You might try changing the initial and final values of the parameters to see if the algorithms are *stable* to these changes. As you form your comments, recall our two competing goals for these algorithms:

- Estimate the values of the actual payoffs (more accurately, the mean payout for each machine).
- Maximize our rewards!

3.3.2 A Summary of Reinforcement Learning

We looked in depth at a basic problem of unsupervised learning- That of trying to find the best winning slot machine in a bank of many. This problem was unsupervised because, although we got rewards or punishments by winning or losing money, we did not know at the beginning of the problem what those payoffs would be. That is, there was no expert available to tell us if we were doing something correctly or not, *we had to infer correct behavior from directly playing the machines¿*

We also saw that to solve this problem, we had to do a lot of *trial and error* learning- that's typical in unsupervised learning. Because an expert is not there to tell us the operating parameters, we have to explore and find them out for ourselves.

We learned some techniques for translating learning theory into mathematics, and in the process, we learned some commands in Matlab. We don't expect you to be an expert programmer - this should be a fairly gentle introduction to programming. At this stage, you should be able to read some Matlab code and interpret the output of an algorithm. Later on, we'll give you more opportunities to produce your own pieces of code.

In summary, we looked at the greedy algorithm, the $\epsilon-$greedy algorithm, the softmax strategy, and the pursuit strategy. You might consider how closely (if at all) these algorithms would reproduce human or animal behavior if given the same task.

There are many more topics in Reinforcement Learning to consider, we presented only a short introduction to the topic. The interested reader should go and find out more!

For us, we now change gears slightly- We'll look at learning from the cellular point of view (versus the holistic behavioral point of view of this section), and we'll focus on a supervised learning problem.

3.4 Donald Olding Hebb and Hebbian Learning

D.O. Hebb (1904-1985) was a physiological psychologist at McGill University. We will discuss his influence on machine learning, but he also did some very interesting experiments to determine how the brain functions. Among the research[3]:

- Evidence for the plasticity of the brain:

> Much to Hebb's amazement, he found that even after substantial loss of tissue from the frontal lobes of the brain, there was no loss in intelligence, and in some cases, he even detected a gain in intelligence[4].

[3] As usual, these footnote sketches are somewhat oversimplified. Read about Hebb's work from his writings, especially "The Organization of Behavior" (1949), and his autobiography in 1980 "A History of Psychology through Autobiography", vol VII

[4] An Introduction to the Theories of Learning, p. 394

- "Monkey Horror" in chimpanzees.
- Sensory Deprivation experiments: Sensory input is vital for brain functioning. Over time without sensory stimulation, subjects hallucinate and their personalities begin to break down.

In Hebb's view, learning could be described physiologically. That is, there is a physical change in the nervous system to accommodate learning, and that change is summarized by what we now call Hebb's postulate (from his 1949 book):

> When an axon of cell A is near enough to excite a cell B and repeatedly takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

As with many named theorems and postulates, this was not an idea that was completely new, but he does give the postulate in a form that can be used as a basis for machine learning. Here are some questions you might think about:

1. Hebb's postulate describes a strengthing or weakening of connections. How is this biologically or chemically done? What exactly is that "physical change"?
2. If this is the basis for learning, is it also the basis of addiction? Friendship? We can't answer these questions, but they are interesting to consider philosophically.
3. The postulate does not give any consideration to *feedback*. If the action causes pain, will the neuron connections still be strengthened? How does emotion in general mitigate or assist in these constructions? Again, we can't give authoritative answers to these questions.

We'll leave these questions for you to think about, and move into something we can answer. How do you mathematically model Hebb's postulate?

3.4.1 Linear Neural Nets and Hebbian Learning

We will go into the formal details later for defining *neural nets*, but this is a good place to get a feel for what they're all about.

Let us first build a simple model for a neuron. A neuron has three body parts- The Dendrites, which carry information to the Cell Body, the Cell Body, and the Axon, which carries information away from the Cell Body.

Multiple signals come in to the cell body from the dendrites. Mathematically, we will assume they all arrive at the same time, and the action of the dendrites (or the arrival site of the cell body) is that each signal is changed by the physiology of the cell. That is, if x_i is information along dendrite i, arrival at the cell body changes it to $w_i x_i$, where w_i is some real value. Next, the cell body collates this information by summing these signals together. So far, then, this action is simply a dot product of the vector of w's (the *weights*) to the signal x. For the purposes of this section, we will assume no further processing.

If the signal is passed to a *cell assembly*, or group of neurons, then the overall action of the group is simply a linear mapping:

$$\mathbf{x} \to W\mathbf{x} = \mathbf{y}$$

If we have k neurons and $\mathbf{x}$ is a vector in $\mathbb{R}^n$, then W is a $k \times n$ matrix, and each row corresponds to a signal neuron's weights (that is, W_{ij} refers to the weight taking x_j to neuron i).

A further assumption will be to specify the mathematical realization of Hebb's postulate. Suppose that we present the network with a pattern, $\mathbf{x} \in \mathbb{R}^n$, and it outputs a pattern, $\mathbf{y} \in \mathbb{R}^k$. In terms of each weight, we see that

$$\boxed{W_{ij} \text{ connects the } j^{\text{th}} \text{ value of the input to the } i^{\text{th}} \text{ value of the output.}}$$

Thus we might take the following as Hebb's Rule. The change in the weight connecting the j^{th} input to the i^{th} cell is given by:

$$\Delta W_{ij} = \alpha y_i x_j$$

where α is called **the learning rate**. If both x_j and y_i match in sign, then W_{ij} becomes larger. If there is a mismatch in sign, W_{ij} gets smaller. This is the *unsupervised Hebbian rule.* Let's take a closer look at this using some linear algebra. Let us suppose we have n inputs to the network, and m outputs. Then W is $m \times n$, with $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$.

Note that if we compute the outer product, $\mathbf{y}\mathbf{x}^T$, we get:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} [x_1, x_2, \ldots, x_n] = \begin{bmatrix} y_1x_1 & y_1x_2 & \ldots & y_1x_n \\ y_2x_1 & y_2x_2 & \ldots & y_2x_n \\ \vdots & & & \vdots \\ y_mx_1 & y_mx_2 \ldots & y_mx_n & \end{bmatrix}$$

You should verify that in this case, we can compactly write Hebb's rule as:

$$W_{\text{new}} = W + \alpha\mathbf{y}\mathbf{x}^T$$

and that this change is valid for a single $\mathbf{x}, \mathbf{y}$ stimulus-response pair. If we define W_0 to be the initial weight matrix (we could initialize it randomly, for example), then the update rule becomes:

$$W_1 = W_0 + \alpha\mathbf{y}\mathbf{x}^T = W_0 + \alpha W_0\mathbf{x}\mathbf{x}^T = W_0\left(I_{n\times n} + \alpha\mathbf{x}\mathbf{x}^T\right)$$

EXERCISES:

1. Write W_n in terms of W_0 using the previous formula as a starting point.

2. Show that, if λ_i, $\mathbf{v}_i$ is the eigenvalue and eigenvector of a matrix A, then $(1 + \beta\lambda_i)$ and $\mathbf{v}_i$ is an eigenvalue and eigenvector of $(I + \beta A)$.

3. Let the matrix $A = \mathbf{x}\mathbf{x}^T$, where $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{x} \neq 0$. If $\mathbf{v} \in \mathbb{R}^n$, show that $A\mathbf{v}$ is a scalar multiple of x. (This shows that the dimension of the columnspace of $\mathbf{x}\mathbf{x}^T$ is 1)

4. Same matrix A as in the previous exercise. Show that one eigenvalue is $\|x\|^2$ (Hint: The eigenvector is $\mathbf{x}$).

5. We'll state the following without proof for now: If A is $n \times n$ and symmetric, and the columnspace of A has dimension 1, then there is exactly one nonzero eigenvalue. Use this, together with the previous exercises to compute the eigenvalues of $I + \alpha\mathbf{x}\mathbf{x}^T$.

6. There is a theorem that says that if the eigenvalues satisfy $|\lambda_i| \leq 1$, then the elements of A^n will converge (otherwise, the elements of A^n will diverge).

   Given our previous computation, will Hebb's rule give convergence?

Somehow, we want to take feedback into account so that we can use Hebb's rule in supervised learning problems.

Let $\mathbf{t}$ be the target (or desired) value for the input $\mathbf{x}$. In the supervised Hebbian rule, we need to update the weights based on what we want the network to do, rather than what the network is already doing:

$$\Delta W_{ij} = \alpha t_j x_i$$

There is still something unsatisfying here- When should we stop the training? It seems like the weights could diverge as training progresses, and furthermore, we're not taking the actual outputs of the network into account. Heuristically, we would like for the update to go to zero as the target values approach the network outputs. This leads us to our final modification of our basic Hebb rule, and is called the Widrow-Hoff learning rule[5]:

$$\Delta W_{ij} = \alpha(t_j - y_j)x_i$$

If we put this in matrix form, the learning rule becomes:

$$W^{\text{new}} = W^{\text{old}} + \alpha\left(\mathbf{t} - \mathbf{y}\right)\mathbf{x}^T$$

where $(\mathbf{x}, \mathbf{t})$ is a desired input-output relation, and $\mathbf{y} = W\mathbf{x}$.

Additionally, sometimes it is appropriate to add a bias term so that the network has the output:

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}$$

The bias update is similar to the previous update:

$$\mathbf{b}^{\text{new}} = \mathbf{b}^{\text{old}} + \alpha\left(\mathbf{t} - \mathbf{y}\right)$$

[5] Also goes by the names Least Mean Squares rule, and the delta rule.

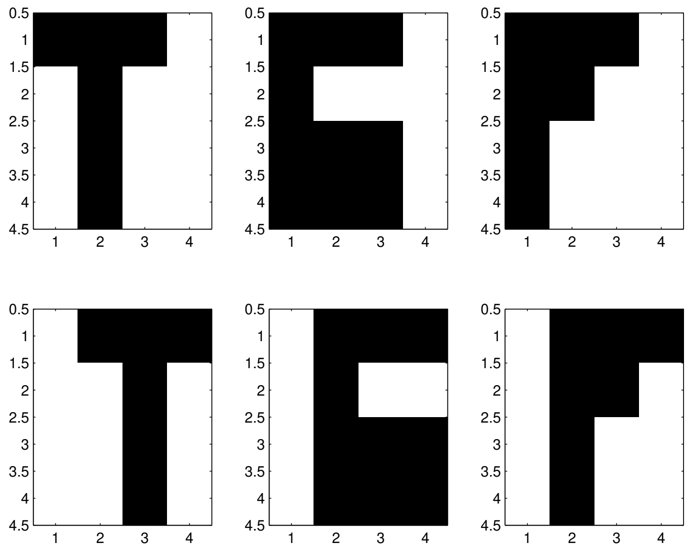

Figure 3.2: The inputs to our linear associative memory model: Three letters, T, G, H, where we have two samples of each letter, and each letter is defined by a 4×4 grid of numbers. We'll be associating T with -60, G with 0, and H with 60.

3.4.2 Example: Associative Memory

Here we will reproduce an experiment by Widrow and Hoff[6] who built an actual machine to do this (we'll do a computer simulation).

We'll have three letters as input, T, G and F. We'll associate these letters to the numbers $-60, 0, 60$ respectively. We want our network to perform the association using the Widrow-Hoff learning rule.

The letters will be defined by 4×4 arrays of numbers, where 1 corresponds to the color black, and -1 corresponds to the color white. In this example, we'll have two samples of each letter, as shown in Figure 3.2.

Implementation:

- First, we process the input data. Rather than working with 4×4 grids, we concatenate the columns to work with vectors in $\mathbb{R}^{16}$. Thus, we have 6 domain data points in $\mathbb{R}^{16}$, two samples of each letter. Construct range points so that they correspond with the letters.

- We'll use an $\alpha = 0.03$.

- We'll take several passes through all the data points.

[6]See "Adaptive Switching Circuits" by B. Widrow and M.E. Hoff, in 1960 IRE WESCON Convention Record, New York: IRE, Part 4, p. 96-104. You might find reprints also on the internet.

- To measure the error, after each pass through the data, we'll put each letter through the function to get an output value. We'll take the square of the difference between that and the desired value. We'll take the sum of the errors squared for those six samples to be the measure of the error for that pass.

Here is the code we used for this example. Again, be sure to read and understand what the code is doing. A lot of the initial part of the code is just there to get the data read in and plotted.

```
T1=[1 1 1 -1
    -1 1 -1 -1
    -1 1 -1 -1
    -1 1 -1 -1];
T2=[-1 1 1 1
    -1 -1 1 -1
    -1 -1 1 -1
    -1 -1 1 -1];
G1=[1 1 1 -1
    1 -1 -1 -1
    1 1 1 -1
    1 1 1 -1];
G2=[-1 1 1 1
    -1 1 -1 -1
    -1 1 1 1
    -1 1 1 1];
F1=[1 1 1 -1
    1 1 -1 -1
    1 -1 -1 -1
    1 -1 -1 -1];
F2=[-1 1 1 1
    -1 1 1 -1
    -1 1 -1 -1
    -1 1 -1 -1];

gg=colormap(gray);
gg=gg(end:-1:1,:);

subplot(2,3,1)
imagesc(T1)
colormap(gg)
subplot(2,3,2)
imagesc(G1)
subplot(2,3,3)
imagesc(F1)

subplot(2,3,4)
```

```
imagesc(T2)
subplot(2,3,5)
imagesc(G2)
subplot(2,3,6)
imagesc(F2)

%*************************************
%The main code starts here!
%*************************************

X=[T1(:) T2(:) G1(:) G2(:) F1(:) F2(:)];
T=[60 60 0 0 -60 -60];
alpha=0.03;
W=randn(1,16);
b=0;
TotalErr=0;
NumPoints=6;

for k=1:60
    idx=randperm(6);

    for j=1:NumPoints
        ThisOut=W*X(:,idx(j))+b;
        ThisErr=T(idx(j))-ThisOut;
    %Update the weights and biases
        W=W+alpha*ThisErr*X(:,idx(j))';
        b=b+alpha*ThisErr;
    end
    EpochErr(k)=norm((W*X+b*ones(1,6))-T);
end
figure(2)
plot(EpochErr);
```

The plot of the error is shown in Figure 3.3. The horizontal axis counts the number of passes through the data, and the vertical axis gives the sum of the squared errors. Note that after 60 passes, we get very good classification of the letters!

You should put this code into Matlab and reproduce the figures.

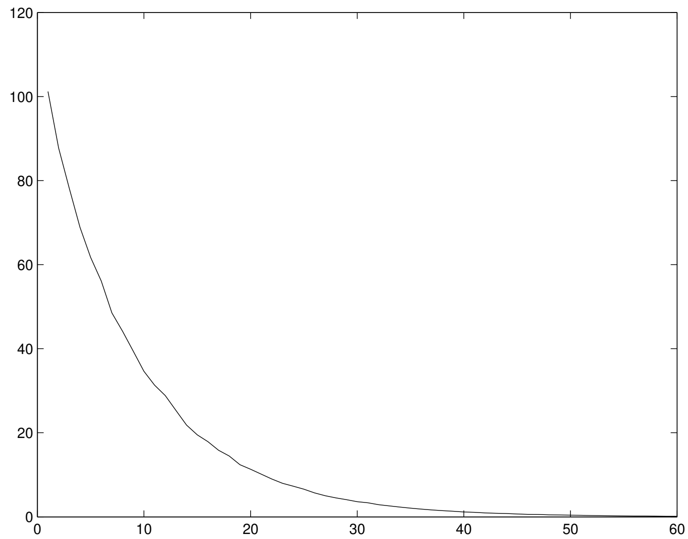

Figure 3.3: The errors for the Widrow-Hoff rule applied to letter recognition (or associative memory. After 60 passes through the data, the associations are very good.

3.4.3 Exercise

Let's put all of this together to solve a pattern classification problem. Suppose we are given the following associations:

Point	Class
$(1, 1)$	1
$(1, 2)$	1
$(2, -1)$	2
$(2, 0)$	2
$(-1, 2)$	3
$(-2, 1)$	3
$(-1, -1)$	4
$(-2, -2)$	4

Graphically, we can see the classes in the plane in Figure 3.4. There are several ways of performing the desired mapping- for example, the outputs could be $1, 2, 3, 4$. But this may have unintended consequences. In this case, the metric between outputs would imply that Class 4 is much farther away from Class 1 than Class 3. A better method may be to take Class 1 to be the vector $[-1, -1]^T$, Class 2 is the vector $[-1, 1]^T$, Class 3 is $[1, -1]^T$, and Class 4 is $[1, 1]^T$. Now the 4 classes are on the vertices of a square.

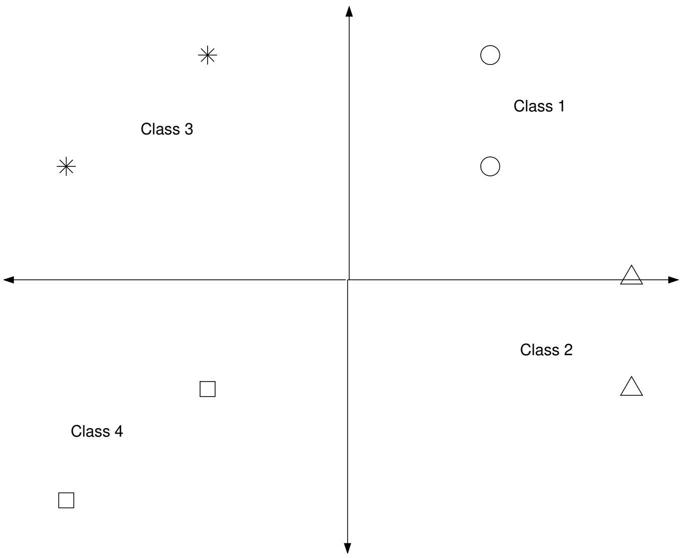

Figure 3.4: Pattern Classification Problem. Each point is a sample of one of the four classes.

Now for the details of the program. First write the inputs as an 2×8 matrix, with a corresponding output matrix that is also 2×8. Parameters that can be placed first will be the maximum number of times through the data N and the learning rate, a, which we will set to 0.04. We can also set an error bound so that we might stop early. Set the initial weights to the 2×2 identity, and the bias vector b to $[1, 1]^T$.

Check your algorithm using the following data: After using the first data point, W and b should have the values:

$$W = \begin{bmatrix} 0.88 & -0.12 \\ -0.12 & 0.88 \end{bmatrix}, \quad b = \begin{bmatrix} 0.88 \\ 0.88 \end{bmatrix}$$

Let the program run until you think it has converged (this is your choice). Then, plot the points $\mathbf{x} = [x, y]^T$ so that $W\mathbf{x} + \mathbf{b} = \mathbf{0}$, which will be the two lines:

$$W_{11}x + W_{12}y + b_1 = 0, \quad W_{21}x + W_{22}y + b_2 = 0$$

These lines form what is called the *decision boundary*.